Curtin University — Department of Computing

UNIX and C Programming 120 (10163)

Semester 2, 2014

Assignment

Due: Week 13 – Monday 27 October, 12:00 noon.

Weight: 20% of the unit mark.

Your task is to create an equation plotter – a program that graphs mathematical functions of x (such as x, $\sin x$, $x^3 + 4x^2$, etc.).

1 Documentation

You must thoroughly document your code using C comments (`/* ... */`).

For each function you define and each datatype you declare (e.g. using `struct` or `typedef`), place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. Collectively, these comments should explain your design. (They are worth substantial marks – see Section 6.)

2 Program Outline

Your program must do the following:

(a) Take two command-line parameters – the names of the input and output files.

(b) Read the contents of the input file, structured as follows:

The first line contains the boundaries of the graph area – real numbers x_{min}, y_{min}, x_{max} and y_{max} – separated by spaces.

Each subsequent line contains three integers, each in the range 0–255, followed by an arithmetic expression. The integers together represent a colour, divided into its red, green and blue components (in that order). Each integer is followed by whitespace, and the expression itself may also contain spaces.

The following is an example input file:

```
-5 -7.5 10 10
0 255 255 x^2 + 2x
192 64 0 -sin x / (cos x - 1)
```

Your program must read the colour values and expressions into a linked list.

(c) Using SDL and `eval.h` (see sections 3 and 4), plot the expressions on the screen in their corresponding colours (assuming "$y =$" in front of each one), on a black background.

Expressions may contain the operators supported by the `eval` function, *as well as* sin, cos, tan, log, exp (i.e. e^x), floor (which rounds down to the nearest integer), and ceiling (which rounds up). Any invalid expression should be skipped over, with one error message only, per expression, shown to the user.

You will need to convert between two kinds of coordinates: pixel coordinates (u, v) and graph coordinates (x, y). Pixel coordinates are integers, where the origin $(0, 0)$ is the top-left-most pixel, and the bottom-right-most pixel is at coordinate $(\text{width} - 1, \text{height} - 1)$ (where "width" and "height" are chosen by your program when it creates the SDL window). By contrast, graph coordinates are real numbers, where x increases from left-to-right, and y increases from bottom-to-top. The input file gives the minimum and maximum graph coordinates (the graph boundaries).

The expressions must be evaluated using graph coordinates (i.e. calculate y given x), but the results must be plotted using pixel coordinates u and v. (Hint: you may actually need to perform the conversion in *both* directions.)

For each expression, your program should draw a continuous line/curve from left to right, except where the value of the expression is undefined or outside the graph boundaries. To do this, it must evaluate one value of y (and thus v) for each possible value of u. If there is a vertical gap between consecutive pixel coordinates (e.g. $(5, 10)$ and $(6, 14)$), then your program will need to plot more pixels in a vertical line (in this case $(6, 11)$, $(6, 12)$ and $(6, 13)$) to fill the gap.

Your program must also plot horizontal and vertical axes, i.e. lines where $x = 0$ and $y = 0$, unless these are outside the range $[x_{\text{min}}, x_{\text{max}}]$ or $[y_{\text{min}}, y_{\text{max}}]$. The axes should be shown in grey (i.e. red = 128, green = 128, blue = 128).

(d) Write to the output file. This will be a CSV (comma-separated value) file, representing a spreadsheet, containing a number of rows and columns. Each line is a row, and values within each row are separated by commas.

The first row contains headings: first "x", then each of the expressions as read from the input file (but no colour codes).

Each subsequent row contains a value of x, followed by the result of each expression for that value of x. There should be one row for each value of x that your program used when plotting the expressions. If your SDL window is 800 pixels wide, there will be 800 values of x, and thus 801 rows in the output file.

The following is a (very small) example of the output file:

```
x,x^2 + 2x,-sin x / (cos x - 1)
-5.00000,15.00000,1.33865
0.00000,0.00000,NaN
5.00000,35.00000,-1.33865
```

```
10.00000,120.00000,-0.29581
```

All numbers should be written with 5 decimal places. If any of the expressions is undefined for a given value of x, write "NaN" instead (meaning not-a-number). If any of the expressions are invalid, write "invalid" for each of its results.

3 SDL (Simple Direct-media Layer)

SDL is a library for (among other things) displaying graphics. To use it, you first need to include the main header file:

```
#include <SDL/SDL.h>
```

You must also tell the linker to use SDL:

```
[user@pc]$ gcc -lSDL file1.o file2.o ...  -o executable
```

(That's a lower-case "L" after the dash.)

The Linux labs in building 314 require some extra setting up. Copy and paste these lines to the end of your `~/.bash_profile` file:

```
export CPATH="${CPATH}:/usr/units/ucp120/include"
export LIBRARY_PATH="${LIBRARY_PATH}:/usr/units/ucp120/lib"
```

Now for the SDL functions and datatypes:

Sint32 and Uint32
: 32-bit signed and unsigned integer types (guaranteed to be 32 bits long, unlike `int` and `unsigned int`).

Uint8
: An 8-bit unsigned integer type.

int SDL_Init(Uint32 flags);
: Initialises SDL, specifying which SDL subsystems to use. Returns 0 for success, or -1 for failure. We're only using the video subsystem, so the argument should be `SDL_INIT_VIDEO` (a pre-defined constant).

SDL_Surface* SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
: Creates a window of the given width, height and bits-per-pixel. For this assignment, an appropriate size would be around 800x800, but you can make it larger if you like (but don't go above typical monitor resolutions). The number of bits per pixel (bpp) should be 32. No flags are necessary, so just pass zero as the fourth argument. The function allocates and returns a new `SDL_Surface` struct, as described below.

SDL_Surface
: A struct type that you can access to plot pixels. Here is a (simplified) declaration:

```
typedef struct {
    SDL_PixelFormat *format;  /* Information on how each pixel is
                                 represented; read only */
    int w, h;                 /* Width and height; read-only */
    void* pixels;             /* Pixel data; read-write */
} SDL_Surface;
```

The `pixels` field points to an area of memory structured like a 2D array of 32-bit values. Each row of the array represents a row of pixels on the screen. Each 32-bit value represents an individual pixel. (We don't need to know what makes up `SDL_PixelFormat`, but we do need to pass it as an argument to `SDL_MapRGB`).

Uint32 SDL_MapRGB(SDL_PixelFormat* format, Uint8 red, Uint8 green, Uint8 blue);
Generates a 32-bit pixel value, based on the surface format and values for red, green and blue.

void SDL_UpdateRect(SDL_Surface* screen, Sint32 x, Sint32 y, Sint32 w, Sint32 h);
Even if you modify `pixels`, the screen is not physically updated until you call `SDL_UpdateRect`. The parameters `x`, `y`, `w` and `h` can all be zero in order to update the whole window.

int SDL_WaitEvent(SDL_Event* event);
Waits (doing nothing) until an "event" occurs. There are many types of events (key presses, mouse movements, etc.), all represented by the `SDL_Event` type. We can use this function to wait until the user presses the close-window button:

```
SDL_Event ev;
...
do
{
    SDL_WaitEvent(&ev);
}
while(ev.type != SDL_QUIT);
```

[Updated on 2014-09-11.]

void SDL_Quit(void);
Shuts down and cleans up after SDL. This should always be called before your program exits.

4 Expression Evaluator

You also have access to `eval.h` and `eval.c`, which contain a function for evaluating mathematical expressions, and some associated datatypes. The details are as follows:

EvalErrorType
An "enum" datatype indicating one of several types of errors that may occur

when evaluating an expression. Enums are not discussed in the lectures until lecture 9, but they are very straightforward. EvalErrorType is a datatype having one of these values:

`ER_NONE` — No error has occurred.
`ER_RESULT_UNDEFINED` — The expression is valid, but the result undefined (e.g. if you divide by zero).
`ER_INVALID_SYMBOL` — The expression is invalid, because it contains a character with no mathematical meaning.
`ER_UNEXPECTED_TOKEN` — The expression contains an invalid sequence of numbers and/or operators.
`ER_UNEXPECTED_END` — The expression ends before it is finished (e.g. if there is a missing close bracket).
`ER_INVALID_OPERATOR` — The expression contains something that looks like a word operator (like "sin", "cos", "log", etc.), but which has no meaning.

EvalError
A struct datatype representing an error and its location in an expression, defined as follows:

```
typedef struct {
    EvalErrorType type;
    char* location;
} EvalError;
```

OperatorEvaluator
A pointer-to-a-function datatype, defined as follows:

```
typedef EvalErrorType (*OperatorEvaluator)(double* result,
                                           char* operatorText);
```

A function of this type evaluates unary operators, such as sin, cos, log, etc. The operator to apply is given by `operatorText`. If valid, the function should perform the requested operation on `*result`, overwriting the original value with the new one.

The function should return `ER_NONE`, `ER_RESULT_UNDEFINED` or `ER_INVALID_OPERATOR` as appropriate.

EvalError eval(double* result, char* expr, double x, OperatorEvaluator opEval);
Evaluates an arithmetic expression (`expr`), placing the result in `*result`. The expression may contain zero or more occurrances of the "x" variable, replaced with the value of the `x` parameter. It may contain the operators `+`, `-`, `*`, `/` and `^`, and brackets `(` and `)`, all with their usual meanings. A range of additional unary operators, such as sin, cos, log, etc. can be provided via an OperatorEvaluator function.

The return value is a struct indicating the details of any error that may occur.

5 Submission

You must submit the following electronically, via the assignment area on Blackboard, inside a single .zip or .tar.gz file (not .rar):

- All your .c and .h files.
- A working Makefile for your program, created by yourself as described in the lectures (*not* automatically generated).

You must also submit a completed, signed "Declaration of Originality" form to the lecturer in person (*not* electronically). Blank forms are available outside the lecturer's office (314.343).

You may make multiple submissions via Blackboard. If all your submissions occur prior to the deadline (i.e. Monday 27 October, 12:00 noon), your newest submission only will be marked. See the Unit Outline for the policy on late submissions.

After submitting, please verify that your submission worked by downloading your submitted files and comparing them to the originals.

6 Mark Allocation

Here is a rough breakdown of how marks will be awarded. The percentages are of the total possible assignment mark:

20% – Using code comments, you have provided good, meaningful explanations of all the files, functions and data structures needed for your *complete* implementation. (If your implementation is incomplete, you may still be awarded some marks for comments relating to functions you haven't yet written.)

20% – For each required piece of functionality, your code is well structured and adheres to practices emphasised in the lectures and practicals. Your code should also be separated into an appropriate set of .c and .h files.

30% – You have correctly implementated the required functionality, according to a visual inspection of your code by the marker.

30% – Your program compiles (with a Makefile), runs and performs the required tasks. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data yourself.

–10% for each practical signoff mark of zero (pro-rata for fractional marks; e.g. –7.5% for each signoff mark of $\frac{1}{4}$).

7 Academic Misconduct – Plagiarism and Collusion

Copying material (from other students, websites or other sources) and presenting it as your own work is **plagiarism**. Even with your own (possibly extensive) modifications, it is still plagiarism.

Exchanging assignment solutions, or parts thereof, with other students is **collusion**.

Engaging in such activities may lead to a grade of ANN (Result Annulled Due to Academic Misconduct) being awarded for the unit, or other penalties. Serious or repeated offences may result in termination or expulsion.

You are expected to understand this at all times, across all your university studies, *with or without* warnings like this.

End of Assignment